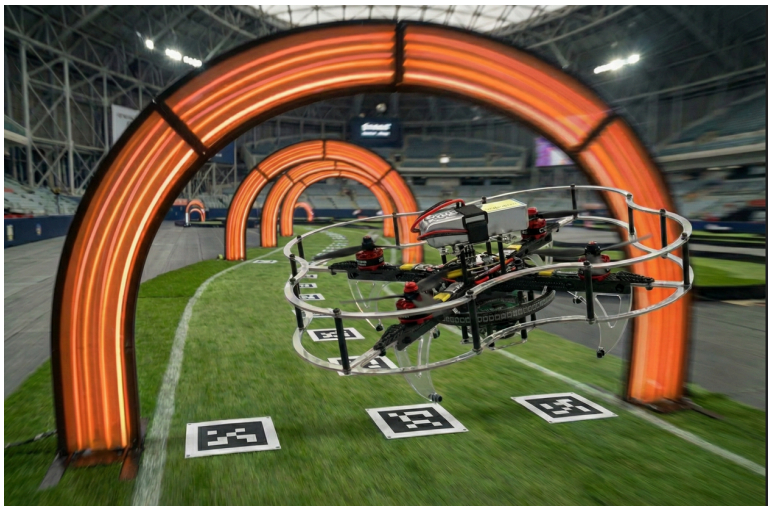
Professional Master of Embedded and Cyber-physical Systems Program,

University of California, Irvine

Capstone Project Report

on

# Autonomous Quadcopter Drone Race




**Submitted By:**

Aneri Hiren Desai (anerid@uci.edu)  Derek Tran (derekt5@uci.edu)

Isaiah Cabugos (icabugos@uci.edu)  Jasera Abdurrashid (jabdurra@uci.edu)

Rohankumar Barouliya (rbarouli@uci.edu)

**Guided by:**

Prof. Yasser Shoukry



**Supported by:**

Resilient Cyber-Physical Systems Lab, UC Irvine

# 1. Project Overview

This project develops a fully autonomous mobility stack for mid-sized quadrotor drones designed for high-speed racing. Using COEX Clover drones equipped with onboard sensors and an Arducam 100 fps mono global shutter camera, the system enables vision-based navigation through multiple race checkpoints. Implemented in the Robot Operating System (ROS), the autonomy stack integrates localization, perception, path planning, and control. Initially validated using a ViCON motion tracking system, the setup now achieves fully onboard operation using ArUco marker–based localization and YOLO-based checkpoint detection.

The multi-modal control layer combines a Model Predictive Controller (MPC) for precise trajectory tracking and a Reinforcement Learning (RL) model for agile decision-making. The RL racing policy, trained for 140 million steps across lightweight simulation and Gazebo, achieves average speeds of 2.85 m/s with peaks of 4.16 m/s. A neural network–trained MPC is converted to TensorFlow Lite and deployed on a Raspberry Pi 4 for real-time execution. The result is an end-to-end, vision-driven, learning-enabled drone platform capable of fast, agile, and fully autonomous flight.

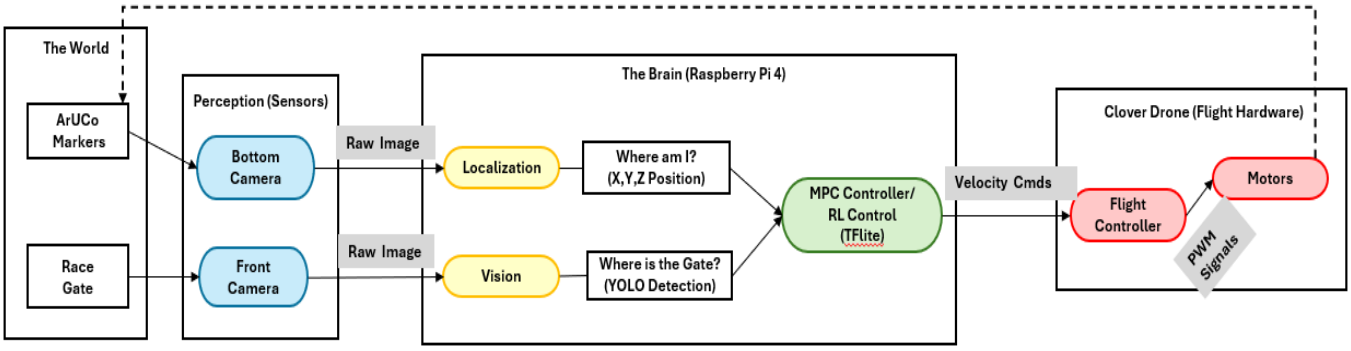



**Fig 1: System architecture**

# 2. Objectives

**Overall Objectives for the Project**

The primary objective of this project was to develop a fully autonomous quadrotor racing system capable of flying through complex tracks at high speeds without external infrastructure. To achieve this, the project aimed to integrate localization, perception, planning, and control into a cohesive onboard autonomy stack. Additional goals included enabling onboard vision-based perception, implementing dual control strategies for both stable and aggressive flight, and validating performance through a complete sim-to-real pipeline.

**Original Objectives for the Fall Quarter**

- As initially discussed with the advisor, the Fall Quarter objectives focused on incorporating advanced sensing and unified control methods to prepare for early autonomous flight tests. These objectives were:
- Integrate a cutting-edge motion-based camera to enable improved localization and perception.
- Combine MPC and RL-trained models into a single navigation stack for intelligent decision-making.
- Conduct indoor drone race flight tests to validate the integrated system in real environments.

**Updated Objectives for the Fall Quarter**

As the project progressed, objectives were refined based on practicality, hardware constraints, and performance considerations. The updated Fall Quarter objectives became:

- Use traditional image-based techniques—ArUco markers and YOLO detection—for robust localization and perception in indoor environments.
- Implement a multi-modal control system, separating MPC and RL modes depending on mission requirements (stability vs. aggressive racing).
- Perform indoor flight tests for both control models to evaluate sim-to-real transfer, system robustness, and onboard execution.

# 3.Setup Details

**Fig 2: Coex Clover Drone 4.2**

| **Software** | **Hardware** |
|---|---|
| <u>Common</u><br>● ROS Noetic<br><u>Simulation/Model Acquisition</u><br>● Gazebo version 11.15.1<br>● Python 3.9<br>● Tensorflow version 2.13.0<br>● TF-agents version 0.17.0<br>● Numpy version 1.24.3<br>● Nvidia CUDA Development Kit 11.8<br>● Gymnasium version 0.29.1 | ● Coex Clover Drone 4.2<br>● Raspberry Pi 4 Model B<br>● PTK Votik 9497 Servo<br>● Arducam 100fps Mono Global Shutter USB Camera<br>● XYG- Raspberry Pi R Camera<br>● Power Distribution Board<br>● MR30 connector ( for motor speed control)<br>● Gemfan Drone racing gates |

# 4.Standards Used/Considered

- To verify ground truth during the early testing phase, we used the Vicon motion-tracking system, which transmitted the drone's position to a base station via UDP over Wi-Fi. As the project progressed, we transitioned to fully on-board localization, phasing out the Vicon system.
- In order to keep track of changes we have a google drive folder in which all of the project related documents are uploaded.
- Our project's source code and version control are managed through a shared GitHub repository under one team member's account. The link to the repository is included below.

  https://github.com/isaiahcabugos/MECPS-Quadrotor-Drone-Race

# 5. Prototypes

## Vision & Perception

- **Vision-Based Localization Using ArUco Markers**

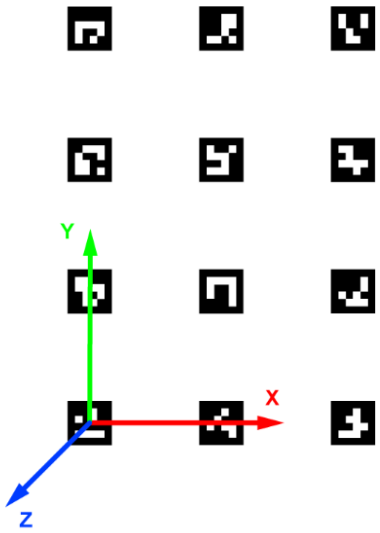



**Fig 3: ArUco Map**

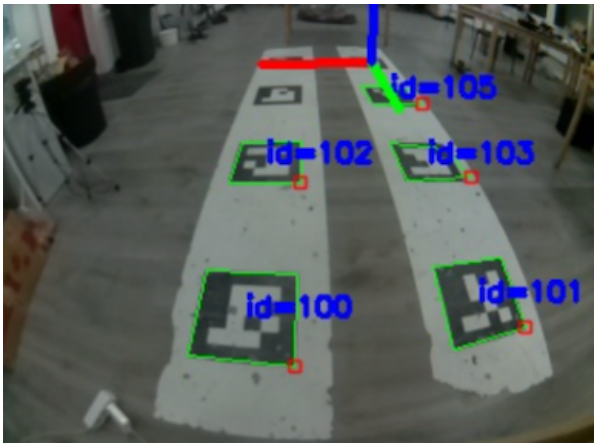



**Fig 4: ArUco Map Lab Setup**

  - The Clover platform includes built-in support for ArUco-based localization, where users can define an aruco_map that specifies the global positions of all markers placed in the environment.
  - For this project, we created a custom ArUco map using markers from a specific ArUco dictionary, assigning each marker ID to a known (X,Y) coordinate within the race arena. This map is stored on the drone and serves as the reference frame for localization.
  - During flight, the Coex Clover's downward-facing camera continuously scans the floor to detect these markers.
  - When a marker is recognized, the drone retrieves its corresponding global coordinates from the onboard aruco_map. By combining this reference information with the relative pose estimation from the camera image, the drone computes its precise global position in real time.
  - The resulting pose estimates are then supplied to the controller, enabling accurate navigation and reliable guidance toward each gate in the racecourse.
- **Object Detection Using Custom YOLO Model**

- The script loads a trained YOLO model (.pt weights) to perform real-time object detection on images/video streams or live camera feeds (USB or PiCamera).
- It identifies a specific target class (i.e., a race gate marker or custom tag) based on the model's label set. Below is an example of a specific chair, as race gates are not yet delivered.

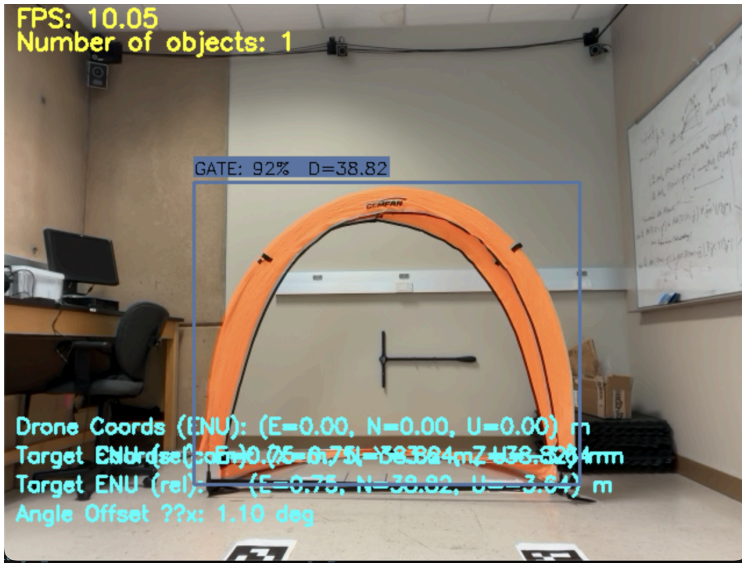



**Fig 5: YOLO-Based Gate Detection**

- **Camera Calibration and Focal Length Estimation**
  - During a calibration phase, the system tracks a known reference object at a known distance and width to auto-compute the camera's focal length using the pinhole camera model:

$$F = (P \times D) / W$$

    where **P** = pixel width, **D** = real distance, **W** = real width.
  - The calibration ensures accurate geometric scaling between image-space pixels and real-world distances.
- **Real-Time Distance Estimation for Localization**
  - After calibration, the code continuously detects the target object and estimates its distance from the drone using the measured bounding box size and computed focal length:

$$D = ( Wref \times f ) / Ppixels$$

  - This pixel-to-distance conversion provides the drone's perception module with range information, enabling localization and navigation decisions (e.g., approaching, centering, or avoiding gates/markers collisions).
  - Below is the Yolo x Distance measurement implementation result:

At calibration object placed 45 inches away from camera, and it's width was 27.5 inches which gave focal length of 477.82px. Then varying the distance as can be seen in the image below at the top of box detection it is showing the distance when object was moved away from camera i.e. 61.40 inches. And these distances estimated by the algorithm are tested & verified by physically measuring the exact distance using a measuring tape.

- **Jetson-Based Real-Time Deployment & Optimization**
  - The YOLO with distance estimation pipeline runs fully on a Jetson Nano (CPU) and streams only the processed target coordinates to the Raspberry Pi and from it to flight controller, reducing onboard load on the drone stack.
  - To achieve smooth real-time performance on the Jetson, we tuned several parameters in the YOLO script:
    - **Frame skipping** where we are processing every 3rd frame so YOLO runs at a manageable rate while the camera still streams at full FPS.
    - **Limiting detections per frame**(i.e. max_det = 5) to avoid expensive post-processing when many objects exist in the scene.
    - **Using a lightweight YOLO model** (my_model_v5n.pt) optimized for embedded hardware instead of a heavier backbone.
    - Fixed CPU device selection (device="cpu") to **match the available runtime environment** and avoid GPU driver issues on this Jetson setup.

After these optimizations, the Jetson reliably delivers stable ENU-relative target coordinates (E_rel, N_rel, U_rel) at real-time rates, which are then sent over UART to the Raspberry Pi for high-level flight control.

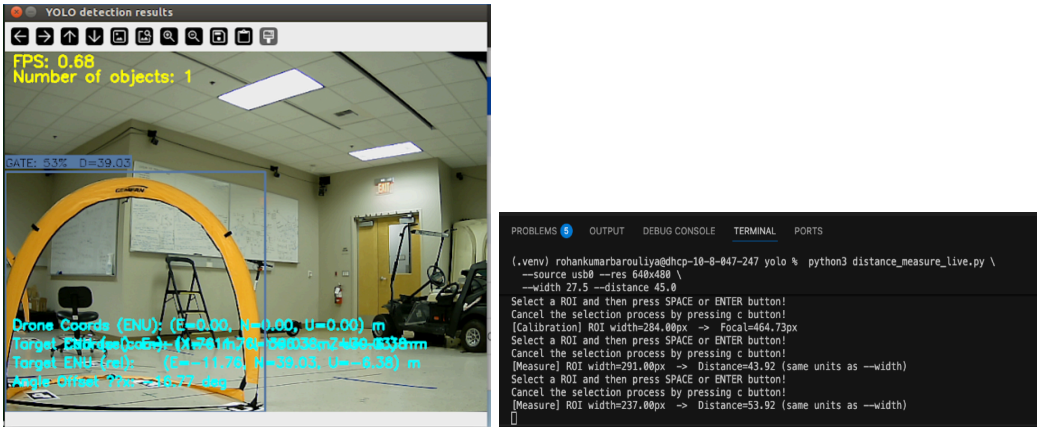



**Fig 6: Gate Detection With Estimated Distance**

# Multi-modal Control

## Model Predictive Control (MPC) Development

Model Predictive Control (MPC) is an optimization-based control strategy that predicts the drone's future motion using a dynamic model and computes the optimal control inputs over a short time horizon. Unlike traditional PID control, MPC anticipates upcoming turns, handles system constraints, and selects actions that minimize a defined cost (e.g., error, energy, or time). These capabilities make MPC particularly well-suited for high-speed autonomous drone racing.

The MPC module for this project was developed through a staged prototyping process, beginning with fundamental altitude control and gradually expanding into full 3D racing maneuvers. The goal was to create a control system capable of high-speed racing while ensuring stability, physical constraint handling, and predictive behavior—features essential for autonomous drone navigation.

- **Stage 1: Z-Axis MPC**
    - The initial prototype focused solely on **altitude control**, allowing us to understand MPC behavior in its simplest form. This stage modeled the drone as a 1D vertical system with height and vertical velocity as states and thrust as the input. Building this version helped the team become familiar with:
        - State evolution using linear dynamics
        - Prediction horizon selection
        - Cost weight tuning
        - Constraint handling
        - Solver speed and numerical stability
    - This minimal version served as a foundation and verified that MPC could stabilize and track altitude setpoints reliably in both simulation and initial hardware tests.

- **Stage 2.1: Full 3D MPC**
    - After validating the vertical controller, the MPC was extended to a full 3D dynamic model, incorporating horizontal motion (X, Y) alongside Z-axis control. This transition revealed important complexities:
        - Coupled system behavior between X, Y, and Z
        - Cross-axis interaction and stability challenges
        - The need for smoother predictions to avoid oscillations
        - Sensitivity to constraint tuning and actuator limits
    - The 3D MPC prototype successfully enabled flight through linear trajectory in simulation and provided a baseline for path tracking.
    - While the 3D MPC prototype performed reliably in Gazebo and successfully tracked linear trajectories in simulation, real-world testing revealed clear gaps between simulation and onboard execution. The drone exhibited oscillatory behavior around checkpoints instead of converging smoothly. This issue was primarily caused by noisy and delayed state estimates, as the get_telemetry() service introduced significant latency during flight. The combination of sensor noise, slow state updates, and the sim-to-real mismatch reduced the controller's ability to react quickly to deviations, causing oscillations and degraded trajectory tracking performance.

- **Stage 2.2: 3D MPC -Simple Turns**
    - After achieving stable linear-path tracking in simulation, the next step was to extend the 3D MPC to handle gentle or single-axis turning trajectories. In Gazebo, the controller successfully predicted the curved path and generated smooth corrective actions using its look-ahead optimization behavior. This stage introduced new challenges such as balancing cross-axis coupling, tuning curvature-related cost weights, and ensuring prediction smoothness during direction changes.
    - At this stage, development remains simulation-focused, as the team is still refining the linear-trajectory MPC for real-world. Real-time testing of turning trajectories is planned as part of future work, once stable localization and streamed state estimation are fully implemented.

- **Stage 2.3: 3D MPC Complex Maneuvers**

- The final MPC extension targets high-speed, multi-axis racing maneuvers, such as S-curves, tight cornering, and rapid direction changes. Initial experiments in simulation demonstrated promising behavior, with MPC leveraging its predictive horizon to smooth trajectories and anticipate aggressive turns. However, this stage is still in progress, with ongoing tuning of horizon lengths, curvature handling, and cost function design to achieve consistent high-speed performance.
- Real-world testing for complex maneuvers remains part of the future scope, as it depends on achieving stable performance in both linear and simple-turn scenarios and improving the onboard state-estimation pipeline. Once these prerequisites are met, the complex-maneuver MPC will move toward hardware validation.

- **MPC Drawback & Solution**
  - Although the MPC-based prototype successfully demonstrated accurate predictive control and optimal racing behavior in simulation, it became clear that full MPC could not run fast enough on the Raspberry Pi used on the drone.
  - Given the computational limitations of running full MPC onboard the Raspberry Pi, the team adopted a **Teacher–Student framework** to retain MPC's intelligence while achieving real-time performance.

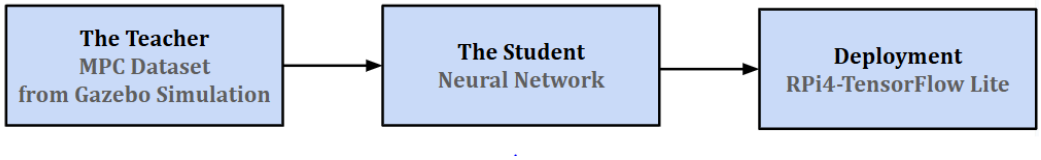



**Fig 7: MPC Deployment Strategy**

  - In this approach, the full MPC controller serves as the "teacher", operating within a high-fidelity simulation environment to generate large amounts of optimal flight data. These datasets include state trajectories, predicted future paths, and the corresponding control inputs computed by the MPC at each step.
  - A **neural network** is then trained as the **"student"** to learn the mapping from the drone's current state to the control commands produced by the teacher. By imitating MPC behavior across a wide range of simulated scenarios, the student model effectively internalizes the teacher's predictive and optimal decision-making abilities.

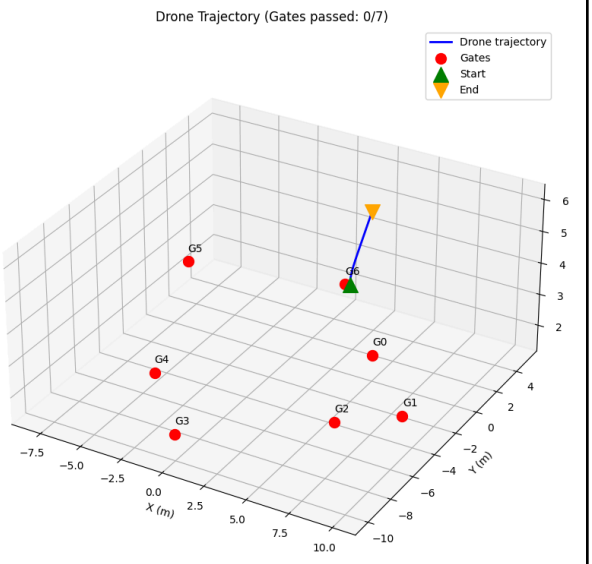
- Once trained, the neural network is converted into a **TensorFlow Lite (TFLite)** model, making it lightweight enough to run efficiently on the Raspberry Pi at **30–40 Hz**, far faster than the 80–120 ms latency of solving MPC directly. This deployment strategy preserves the key advantages of MPC—such as constraint handling, smooth trajectories, and predictive behavior—while enabling fast, reliable, and fully onboard control suitable for real-world drone racing.

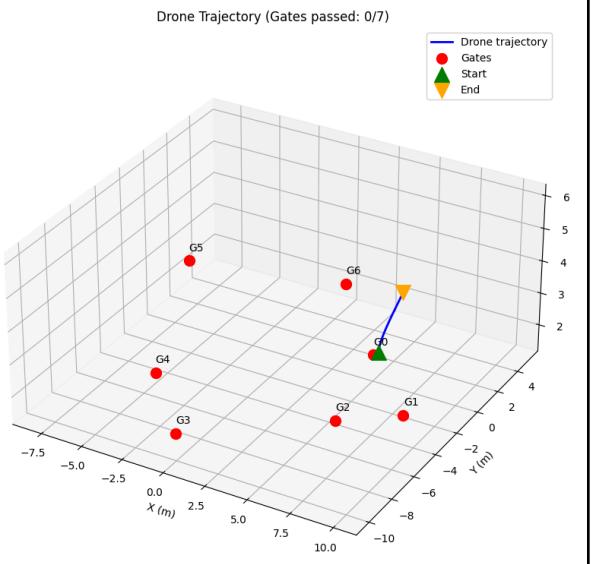
## RL Model

- **RL Racing Model v0.1:**
  - This iteration had numerous bugs in both the physical simulation and the handling of the RL model. Not only were the physics of the drone not accurately modeled, the rewards were not calculated correctly. As a result, the model could never learn any meaningful function. The resultant output is only an unstable representation of noise.

| | |
|---|---|
| ```<br>============================================================<br>TRAINING STATUS<br>============================================================<br>Step: 12,000,000 / 20,000,000 (60.0%)<br>Episodes: 1425034 (249062 with rewards)<br>Success Rate: 0.00%<br>Avg Reward: -7.352<br>Last Update: 2025-10-09T22:01:44.366862<br>------------------------------------------------------------<br><br>============================================================<br>FINAL EVALUATION RESULTS (100 episodes)<br>============================================================<br>Success Rate: 0%<br>Average Final Distance: 5.50 m<br>Median Final Distance: 5.43 m<br>============================================================<br>``` | After 12 million steps, the RL model did not learn any meaningful function. Out of 100 episodes, 0% of the episodes passed through a single gate (passing through a gate counts as a success). |

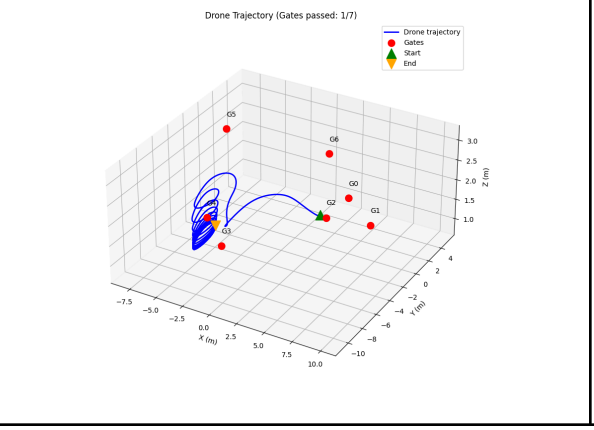
- **RL Racing Model v0.2:** The below images showcase four different flightpaths for the drone starting at random gates. In each image, the model controls the velocities of the drone. The green arrow represents the starting location of the drone, and the yellow arrow represents the drone's location at the termination of the simulation.

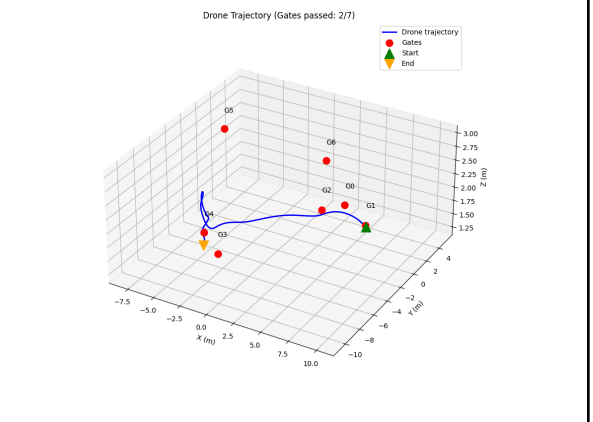



Visualization techniques have been implemented, allowing for a more thorough investigation of the model outcomes. Furthermore, the RL model has been modified to allow flying through multiple gates in succession, rather than a single gate. It is here where a critical bug was found with the body frame and world frame of the drone. Training for 50 million steps resulted in this model. Out of 10 practice simulations, every single drone ended the simulation because of crashing at the ceiling.

- **RL Model v0.3:** These images showcase the same as the above section, except using the newer model.



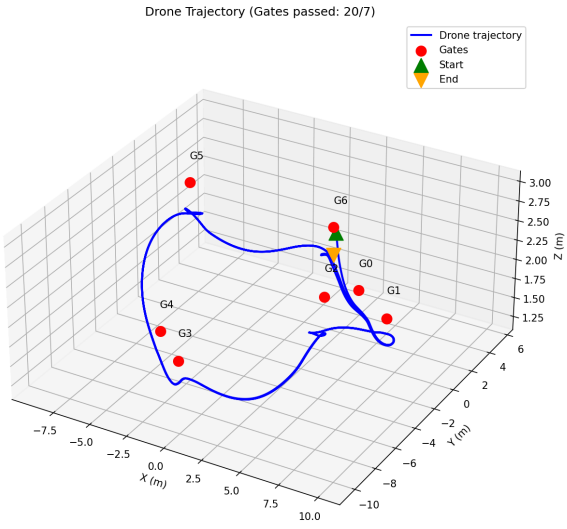



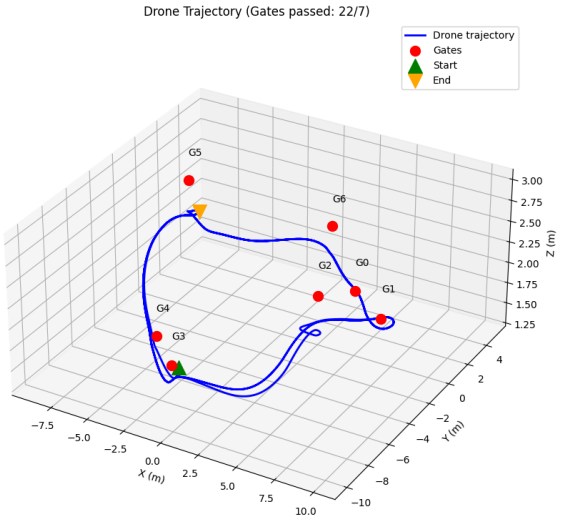

After rectification of an additional bug with the flight control stack of the physics simulation, the model can now demonstrate the ability to maneuver itself in an agile and relatively accurate manner. After 1.5 million steps, it can now fly through multiple gates

in a row. The paper we are basing this model on utilized 100 million steps. With more training, this RL algorithm should be sufficient for our purposes.

- **RL Model v1.0:** The section below details the final iteration of the RL Racing model training architecture.

| | |
|---|---|
| **[Observation Space]:** 30 Parameters<br>**Position (x, y, z):** [3]<br>**Velocity (x, y, z):** [3]<br>**Rotation Matrix:** [9]<br>**Gate Corners (4 corners in x,y,z):** [12]<br>**Previous Action:** [3]<br><br>**[Action Space]:** 3 Parameters<br>**Velocity Command (x, y, z):** [3] | Trained for 100M Steps<br><br>Training System Specifications:<br>CPU: Intel i5-13600K<br>RAM: 32GB DDR4 3600MHz<br>GPU: Nvidia RTX 3050 8GB<br><br>Training Time: ~25 Hours |
| **Rewards Function: Dense Reward**<br><br>**r_progress + r_perc + r_smooth + r_crash**<br><br>**r_progress** = lambda1 * (prev_distance - current_distance)<br><br>**r_perc** = lambda2 * np.exp(-lambda3 * (delta_cam ** 4))<br><br>**r_smooth** = -(lambda4 * np.linalg.norm(self.previous_action)**2 + lambda5 * np.linalg.norm(action_change))<br><br>**r_crash** = 5.0 if crash, 0.0 otherwise. | **lambda1** = 1.0<br>**lambda2** = 0.02<br>**lambda3** = 10.0<br>**lambda4** = 0.0001<br>**lambda5** = 0.0002<br><br>**Episode Timeout:** 1500 Steps<br>**Step Rate:** 0.004s/step<br>**Control Rate:** 0.04s/step<br><br>**Drone Gains and Specifications:**<br>Standard as per official Clover specifications |




In our final iteration for the lightweight custom sim, 100 million steps was able to yield a model which can lap the track multiple times within one training episode. In this simplified training environment, it no longer exhibits oscillatory behaviors. After verifying these suitable behaviors, we transfer the model to Gazebo for more rigorous physics-based testing.

# 6. Experiments/Results:

## Test Setup

- **Simulation (SITL) Setup (Completed):**
  - **Environment: Gazebo** 11.15.1
  - **Middleware: ROS Noetic**
  - **Flight Stack: PX4 SITL**

    - **Drone Model:** A simulated **Covex Clover** drone.
- **Real-World (Physical) Setup (In Progress):**
    - **Environment:** A physical lab space.
    - **Drone Hardware:** The physical **Covex Clover** drone
    - **On-board Computer:** Raspberry Pi 4.
    - **On-board Control Model:** TensorFlow Lite (.tflite) model converted from trained "student" NN.
    - **On-board Localization:** ArUco code detection algorithm

## Test Cases, Objectives, and Analysis

### Phase 1: Simulation (Completed)

**MPC Model**

| | |
|---|---|
| Test Case 1 | 3D MPC "Teacher" Validation |
| Objective | To verify that the baseline 3D MPC controller can stably and accurately navigate the drone through gates in a straight line. |
| Input | • ROS node running the 3D MPC algorithm.<br>• Gazebo world with straight-line gates. |
| Expected Output | The drone flies smoothly through all gates without crashing. |
| Actual Output | Success. The drone performed as expected, validating the "teacher" model. |
| Analysis | The MPC controller is functional and reliable enough to generate training data. |

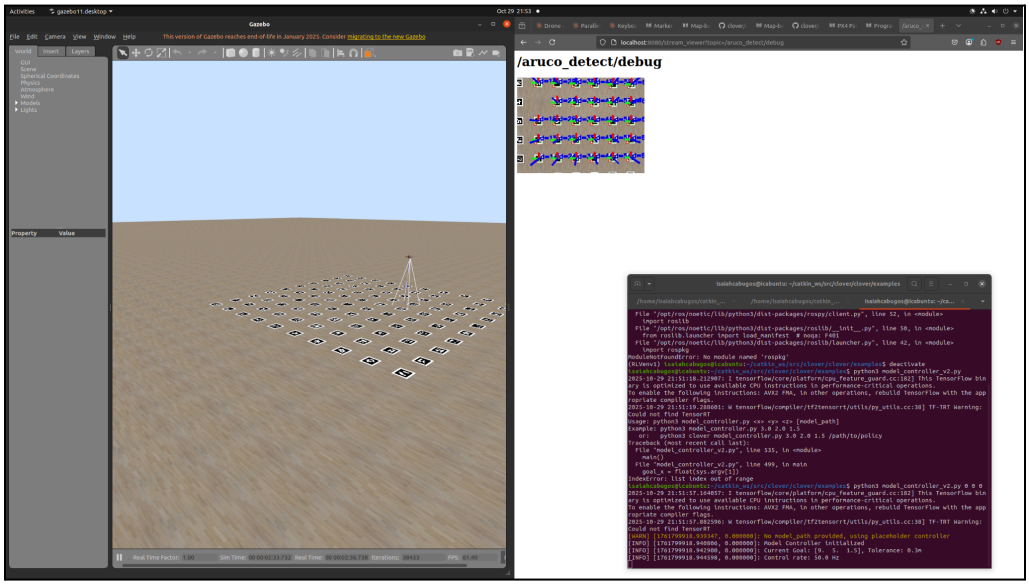
| | |
|---|---|
| Test Case 2 | "Student" NN Model Validation - Small Dataset |
| Objective | To test if a "student" NN trained on a small dataset (10 or 30 simulations) can control the drone. |
| Input | • Trained student_model.h5 from 10/30 sim datasets.<br>• ROS node to run the NN model.<br>• Gazebo world with gates. |
| Expected | The drone flies smoothly through all gates in a straight path, closely |

| | |
|---|---|
| Output | mimicking the MPC. |
| Actual Output | Failure. The drone was unstable and did not accurately pass through the gates in Gazebo Simulation Environment. |
| Analysis | The dataset was too small and lacked sufficient variation. The model failed to generalize and could not learn the control policy. This was a critical finding. |

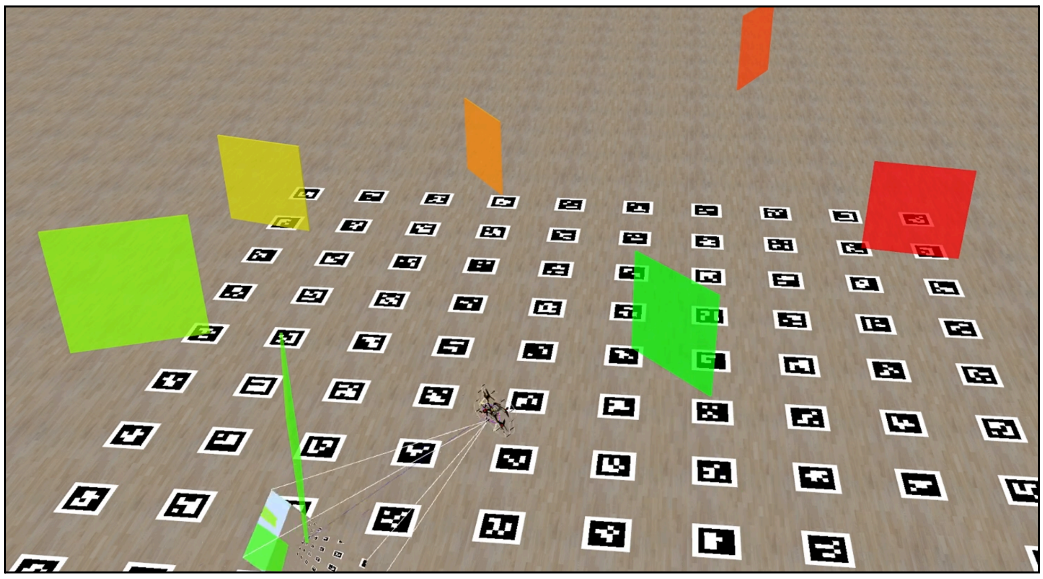
| | |
|---|---|
| Test Case 3 | "Student" NN Model Validation - Large Dataset |
| Objective | To test if a "student" NN trained on a larger dataset (100 or 200 simulations) can control the drone. |
| Input | • Trained student_model.h5 from 100/200 sim datasets.<br>• ROS node to run the NN model.<br>• Gazebo world with gates. |
| Expected Output | The drone flies smoothly through all gates in a straight path, closely mimicking the MPC. |
| Actual Output | Success. The model trained "quite well" and was able to fly the drone through the gates. |
| Analysis | This confirms that 100-200 simulations provide a dataset large enough for the NN to learn the control policy for a linear trajectory. This model is now the candidate for real-world testing. |

**RL Model**

| | |
|---|---|
| Test Case 4 | Simple RL Racing Algorithm Gazebo Simulation |
| Objective | Test if the RL model can suitably control the drone to reach a set goalpoint on a predetermined map and land on a predetermined point. A focus on the drone's on-board localization capabilities. |

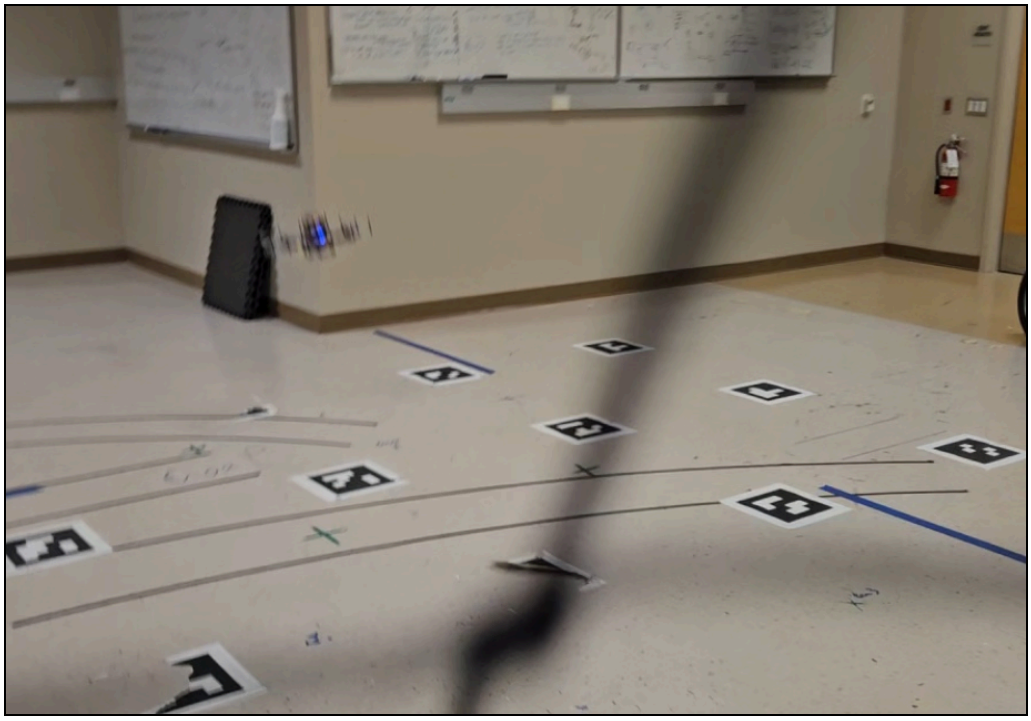
| | |
|---|---|
| Input | • Trained RL model<br>• ROS node to run the model<br>• Gazebo world with ArUco map<br>• Hard-coded goal point |
| Expected Output |  <br>The drone flies to the goalpoint quickly, and then lands. |
| Actual Output | Occasional Success. The model is able to fly quickly to the location, but sometimes misses the goal, at which point it flies off or flies in a circle trying to reach the goal. |
| Analysis | The critical failures from this design show that there must be hard-coded safety measures in the control algorithm. Furthermore, the RL model is not designed with generalization in mind; The racetrack it was trained on is the only racetrack it is able to fly in. This must be kept in mind for the final physical test. |

| | |
|---|---|
| Test Case 5 | Advanced RL Racing Algorithm Gazebo Simulation |
| Objective | Verify if the model is able to fly to multiple gates in succession. Gather residuals based on intended movements by model and actual velocities in Gazebo. |

| | |
|---|---|
| Input | • Complete RL model trained on Residuals<br>• Gazebo world with ArUco map<br>• Hard-coded series of goals |
| Expected Output |  <br>Drone flies between some goals but ultimately encounters some issues from sim differences. |
| Actual Output | Rare success; Drone often exhibits behavior of missing goal and oscillating the miss multiple times. Eventually the drone is able to reach the goal and continue but the overall performance of the model is poor. Difficult to discern average success rate. Sometimes the model is able to reach goals, sometimes the model misses a goal and oscillates, resulting in an eventual crash. |
| Analysis | The minimal improvements compared to the base model indicate that the residual collection and training phase of this project requires additional work. The research paper we based our work upon was able to reach suitable results within 20 million residual-trained steps. We were not able to replicate that, and as such our training process needs a deeper analysis. |

## Phase 2: Real-World Deployment

| | |
|---|---|
| Test Case 6 | Real-World Unit Test: Localization (ArUco) and Navigation |
| Objective | To verify the on-board camera and ArUco detection algorithm can provide accurate, stable, and high-frequency pose data. Test if the drone can additionally navigate itself based on this data. |
| Input | • Physical drone (can be stationary or handheld).<br>• RPi 4 running the ArUco detection node.<br>• Physical ArUco markers placed in the test area. |
| Expected Output | • A ROS topic (e.g., /aruco/pose) publishes the drone's pose.<br>• The pose data must be accurate (compare to physical measurements), stable (low jitter), and fast enough for the control loop.<br>• The drone hovers in a stable manner above one of the ArUco markers. |
| Actual Output |  <br>Drone shows recognition of ArUco markers. |

| | |
|---|---|
| |  <br>Drone seconds before flying into the ceiling. |
| Analysis | The drone has the ability to perceive and localize itself relative to Aruco markers. Occasional catastrophic failures lead to the drone flying in an uncontrolled manner, likely due to LPE. EKF2 will also be tested to see suitability. Physical drone characteristics do affect the stability of the drone, as expected. Future RL algorithms must be designed with this in mind; Collecting residuals is a good start. |

| Test Case 7 | Real-World Unit Test: TFLite Model Performance and Latency |
|---|---|
| Objective | To verify that the .tflite model runs efficiently on the RPi 4 and that the localization and control nodes can run in parallel without overloading the CPU. |
| Input | • RPi 4.<br>• Run the ArUco node (from Test 4).<br>• Run the TFLite inference node (feeding it test data). |
| Expected Output | • Both processes run simultaneously.<br>• htop (or similar) shows CPU and memory usage are at safe levels (e.g., < 80-90%).<br>• TFLite inference speed is high (e.g., 30+ Hz). |

| | |
|---|---|
| Actual Output | Both the TFLite controller and ArUco localization ran simultaneously on the Raspberry Pi 4 without any issues. CPU usage stayed below safe limits, and the TFLite model consistently achieved **30+ Hz** inference speed with no frame drops or delays. |
| Analysis | The test confirms that the onboard system can run localization and the TFLite controller in parallel without overloading the hardware. This demonstrates that the optimized student controller is fast enough for real-time deployment and that ArUco-based localization is reliable on the RPi 4. |

| Test Case 8 | Full System MPC Test |
|---|---|
| Objective | To perform the flight test of the fully integrated, fully on-board system (TFLite controller + ArUco localization). |
| Input | • Physical drone with RPi 4.<br>• TFLite model running on the Pi.<br>• ArUco localization running on the Pi.<br>• A linear race path with two physical gates. |
| Expected Output | The drone autonomously and successfully navigates the entire linear racecourse with two gates. |
| Actual Output | During the linear trajectory test, the drone was unable to maintain a stable path and oscillated around the checkpoints instead of progressing smoothly through the gates. The controller repeatedly over-corrected its position, preventing successful completion of the full race path. |
| Analysis | The instability was primarily caused by the **simulation-to-real gap** and **noisy, delayed state estimates** from the onboard localization pipeline. Because the controller received outdated or jittery position data, its corrective actions were consistently behind, leading to oscillations. These findings indicate that improved state-estimation (streamed telemetry, filtering) and further tuning are required before reliable real-world MPC-based navigation can be achieved. |